Integrating Specification and Documentation in an Object-Oriented Language

[Extended Abstract]

Jie Liang
Department of Computing and Software
McMaster University
Hamilton ON Canada, L8S 4K1
liangj2@mcmaster.ca

Emil Sekerinski
Department of Computing and Software
McMaster University
Hamilton ON Canada, L8S 4K1
emil@mcmaster.ca

1. INTRODUCTION

This paper reports on the integration of specification and documentation features into an object-oriented programming language and its compiler. The goal of this integration is to improve software quality, in particular correctness, extensibility, and maintainability in a uniform and coherent manner. The language taken is Lime, an action-based concurrent object-oriented language developed at McMaster University. The concurrency aspect of Lime is motivated by the observation that concurrency is increasingly used to improve responsiveness of programs. Concurrency in Lime is expressed by attaching *actions* to objects. This eliminates the conceptual distinction between objects and threads. For the theory behind this approach and an implementation scheme the reader is referred to [12].

This paper focuses on features that are being added in order to improve software quality. We argue that specification and documentation means need to be integrated in a programming language. The documentation can be easier kept up-to-date if there is no need to switch between the programing and documentation environments; outdated documentation is a common problem [11]. If specifications are "first-class citizens", then the means for structural checks, composition, reuse, and documentation can be extended to specifications, in addition to offering the possibility for behavioral checks. We argue that for object-oriented programs to support specification, a strict separation of subclassing (code sharing) and subtyping (substitutability) is needed. Such a separation allows each class to serve as a superclass (be reused) or of a supertype (be implemented) and any child class to either inherit the implementation of the parent class, the behavioral specification, or both. Behavioral specifications are expressed by preconditions, postconditions, and invariants. These and other intermediate annotations can be written using quantifiers and other standard mathematical notation, and can be checked at run-time. The associated documentation tool generates a description of the interface of each class that includes the preconditions and postconditions of the methods, the class invariant, the subtype hierarchy, and the subclass hierarchy. Mathematical symbols in the source file are represented by Unicode characters. The compiler, LimeC, generates code for the Java Virtual Machine and the documentation tool, LimeD, generates HTML. The behavioral specifications are embedded in the generated JVM files. When inheriting from a class of a different compilation unit, both LimeC and LimeD extract these specifications from the object files of classes; the source code and separate documentation are not needed.

SAVCBS'04 Newport Beach, California, USA

2. INTEGRATING SPECIFICATIONS

The interface specification languages in the Larch family [13], JML [2, 3] and Eiffel [10] specify the behavior of their modules by Hoare-style correctness assertions.

Design by Contract (DBC), proposed by Meyer for Eiffel [10], is a formal technique for dynamically checking specification violation during run-time. The idea behind DBC is that a class and its client have a "contract" with each other. The client must guarantee certain conditions before calling a method defined by the class, and in return the class guarantees certain properties that will hold after the call. In Eiffel, the contracts are defined by program code, and are translated into executable code by the compiler. Thus, any violation of the contract can be detected immediately during run-time.

The lack of assertions and design by contract features in Java has led to some languages and run-time assertion checking tools, such as Alloy Annotation Language (AAL) [6], Jass [1], and iContract [7]. AAL is a language for annotating Java code based on the Alloy modeling language.

JML, which stands for "Java Modeling Language," is a behavioral interface specification language (BISL) designed to specify Java modules. JML adds assertions to Java by writing them as special comments (/*@ ... @*/ or //@ ...). It is based on the use of preconditions, postconditions and invariants. JML uses Java's expression syntax to write the predicates used in assertions. Java expressions lack some expressiveness that makes more specialized assertion languages convenient for writing behavioral specifications; JML solves this problem by extending Java's expressions with some specification constructs, such as quantifiers.

Lime integrates assertions as programming language constructs, as Eiffel does; for offering a trade-off between checking for correctness and efficiency, assertion checking can be selectively enabled and disabled. Lime offers *class invariants*, *precondition*, and *postconditions* to specify module behavior. In addition, Lime has *assert* statements which specify a constraint on an intermediate state in a method body.

To have more expressiveness for writing behavioral specifications, Lime allows the following constructs in expressions:

- Boolean operators $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$. Note that $\Leftrightarrow$ means the same as $=$ for expressions of type *boolean*; however, $\Leftrightarrow$ has a lower precedence.
- *old e* for referring to a value in the pre-state. It is used in postconditions to indicate an expression whose value is taken from the pre-state of a method call. For example, *old(x + y)* denotes the value of *x + y* evaluated in the pre-state of a method call.
- *result* for referring to the value or object that is being returned by a method. It is used in a method's postcondition. Its type is the return type of the method.
- linear quantifications $* \, x \, | P \bullet E$, where $*$ is a quantifier operator; x is the bound variable; P is the range; E is the body of the quantification; $*$ is $\forall$, $\exists$, Σ, Π, MAX, MIN, or NUM. The range has the form $E_{low} \preceq x \preceq E_{up}$, where $\preceq$ is either $<$ *or* $\leq$.

To make the source code and generated documentation more readable and meaningful, we use a number of mathematical symbols. With the aid of Unicode and UTF-8 encoding, these mathematical symbols can be parsed by the compiler and displayed on any word processor that supports UTF-8 enconding for editing source code and inside generated documentation on web browsers.

Since Java and JVM do not support behavioral specifications, we need a way to store the preconditions, postconditions and invariants in a Java *class* file. When handling inheritance and separate compilation, we only get information about other compilation units from their Java class file. The reason is that in some situation such as using a library class, we may not have the source code. The class invariant, preconditions and postconditions are therefore stored in Java class files as constant strings. They are extracted as constant values from the constant pool by LimeC and LimeD.

3. TYPES AND CLASSES

Inheritance is a language mechanism that allows new object definitions to be based on existing ones. A new class inherits the properties of its parents, and may introduce new properties that extend, modify or defeat its inherited properties. Subtyping and subclassing are conceptually different views of inheritance: Subtyping is related to specification and interface inheritance; subclassing is a mechanism for implementation reuse.

Cook [4] points out that in most strongly-typed object-oriented languages subtyping are subclasses are combined and equated, and inheritance is basically restricted to satisfy the requirements of subtyping. It has been argued that this eliminates several important opportunities for code reuse [8, 10]. Currently, only a few languages, such as POOL-I, Theta, PolyTOIL and Sather, support separating subtyping and subclassing to some degree.

$\tau \;\; extend \;\; \sigma$

1. τ inherits every non private attribute a_σ of σ: $\tau.A \supseteq \sigma.A$.
2. For any non private method m_σ of σ there is a corresponding method m_τ of τ, such that
   - m_τ has m_σ's signature: $m_\tau.Sig = m_\sigma.Sig$.
   - m_τ has m_σ's implementation: $m_\tau.Imp = m_\sigma.Imp$.

Figure 1: Definition of *extend*

$\tau \;\; implement \;\; \sigma_1, \sigma_2, ..., \sigma_n$

1. τ preserves invariants of all supertypes $\sigma_1, \sigma_2, ..., \sigma_n$: $\tau.I \Rightarrow \bigwedge_{i=1}^{n} \sigma_i.I$.
2. τ inherits all non private attributes from all supertypes $\sigma_1, \sigma_2, ..., \sigma_n$:
   $\tau.A \supseteq \bigcup_{i=1}^{n} \sigma_i.A$.
3. For any non private method m_{σ_i} of each supertype σ_i there is a corresponding method m_τ of τ, such that
   - m_τ has m_{σ_i}'s signature ($m_\tau.Sig = m_{\sigma_i}.Sig$).
   - m_τ weakens preconditions: $m_\tau.Pre \Leftarrow \bigvee_{i=1}^{n}(m_{\sigma_i} \in \sigma_i.M \wedge m_{\sigma_i}.Pre)$.
   - m_τ strengthens postconditions: $m_\tau.Post \Rightarrow \bigwedge_{i=1}^{n}(m_{\sigma_i} \in \sigma_i.M \Rightarrow m_{\sigma_i}.Post)$.

Figure 2: Definition of *implement*

Syntactic subtyping can be extended to behavioral subtyping. The essence of behavioral subtyping is summarized by Liskov and Wing's subtype requirement [9]:

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

We propose an inheritance mechanism that strictly separates subclassing from subtyping and makes inheritance more flexible. Any class in Lime can act as a superclass or a supertype. A class contains a syntactic interface, the specified behavior, and an implementation. A child class has the choice of inheriting either the behavioral specification, the implementation, or both.

A Lime class definition consists of a class invariant (*I*), a set of attributes (*A*) and a set of methods (*M*). We model a class as a triple $\langle I, A, M \rangle$. A method is composed of a signature (*Sig*), behavioral specification and implementation (*Imp*). The method signature includes name, access, return and parameters' types. The behavioral specification consists of a precondition (*Pre*) and postcondition (*Post*). The implementation is the source code of the method body. We model a method as a quadruple $\langle Sig, Pre, Post, Imp \rangle$.

Lime uses the *extend* clause to handle single subclassing (Figure 1) and the *implement* clause to handle multi-subtyping (Figure 2). The case of combined subclassing and subtyping is expressed by the *inherit* clause. The subtyping definitions follows that of [9]; JML uses the generalization of [5]. For example, the class header

class Sub extend Sup2 implement Sup1

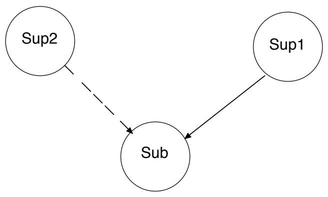

Figure 3: Inheritance graph in Lime

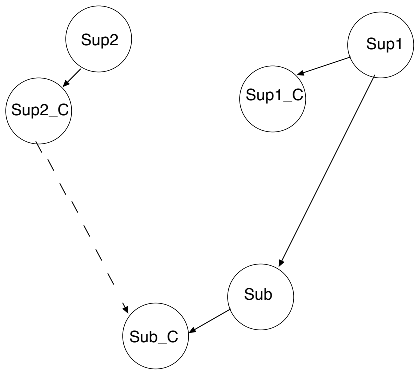

Figure 4: Inheritance graph of generated Java classes

builds an inheritance relation shown by the inheritance graph in Figure 3. Solid and dashed arcs are used to represent subtype and subclass relationships, respectively.

We sketch how the inheritance relationship is implemented in the generated executable Java class file. Java supports single inheritance of classes and multiple inheritance of interfaces that can only contain method signatures and constant static variables. For each Lime source file, we generate two Java class files. One stores a Java class that contains all the information in the original Lime file, and its name ends with "*_C*", the other stores a Java interface that still uses its original name. The graph in Figure 4 shows the inheritance relationship among the generated Java classes. In Java, it would be legal to assign an instance of *Sub_C* to a variable declared as of type *Sup1* or *Sup2*. According to our definition of *extend*, *implement* and *inherit*, class *Sub* is *Sup1*'s subclass, not subtype. The compiler checks whether the variable being assigned is of a supertype of the instance's class. From the inheritance graph view, this amounts to checking whether there exists a path that is composed of all solid arcs between two types.

In the following example, class *Polygon* is only a subclass of *Rectangle*, not a subtype. It can reuse the code in class *Rectangle* such as method *boundingBox*. It also overrides method *move* and *area*. Quantifier $\forall$ is used for specifying the behavior of method *move*. In the initialization, *MAX* and *MIN* are used for calculation.

```
abstract class Shape
  protected attr x, y : integer;
  public abstract method boundingBox : Rectangle;
  public method area : integer
     return 0
  public method move (dx, dy : integer)
    begin
      x : = x + dx;
      y := y + dy
    end
  initialization(x, y : integer)
    begin
     self.x := x;
     self.y := y
  end
end

class Rectangle inherit Shape
  protected attr w, h : integer;
  public method boundingBox : Rectangle
     return new Rectangle(x, y, w, h)
  public method area : integer
     return w * h
  initialization(x, y, w, h : integer)
    begin
      super.initialization(x, y);
      self.w := w;
      self.h := h
    end
end

class Polygon extend Rectangle implement Shape
  protected attr i, n : integer;
  attr Xs : array of integer;
  attr Ys : array of integer;
  invariant n > 0
  initialization(Xs, Ys : array of integer, n: integer)
    begin
      self.n := n;
      x := Xs[0];
      y := Ys[0];
      w := $(MAX\ i \mid 0 \leq i < n \bullet Xs[i]) -$
                 $(MIN\ i \mid 0 \leq i < n \bullet Xs[i]);$
      h := $(MAX\ i \mid 0 \leq i < n \bullet Ys[i]) -$
                 $(MIN\ i \mid 0 \leq i < n \bullet Ys[i]);$
      i : = 0;
      while i < n-1 do
        self.Xs[i], self.Ys[i] := Xs[i+1], Ys[i+1];
    end
  public method move (dx, dy : integer)
    post $\forall\ i \mid 0 \leq i < n - 1 \bullet (dx =$
         old Xs[i] - Xs[i]) $\wedge$ (dy = old Ys[i] -Ys[i])
    begin
      super.move(dx, dy);
      i := 0;
      while i < n - 1 do
        Xs[i], Ys[i] := Xs[i] - dx, Ys[i] - dy
    end
  public method area : integer
    ...
   end
end
```

4. DOCUMENTATION GENERATION

Lime's support for automatic documentation generation was influenced by early work on literate programming and documentation system like *Javadoc* and *Doxygen*. Both Javadoc and Doxygen generate on-line interface documentation in HTML format. The design for LimeD is along those lines:

- LimeD generates documentation directly from the source code;
- LimeD provides a behavioral interface specification, not only a syntactic interface;
- LimeD shows the subclass and subtype hierarchies.

For a project, LimeD generates a summary page and a page for each individual class. For quickly accessing class documentation, a list with linked indices for all classes is generated and acts as a navigation menu. The documentation of each individual class starts with the class description extracted from the documentation comment in the source file. Documentation comments can contain embedded HTML code. The document may contain the following parts:

- **Class Invariant** with the invariant defined in the current class and the invariants inherited from all supertypes. The inherited invariants are conjoined to generate a single expression. All the information is extracted from the current class and from the Java class files of all supertypes.
- **Class Hierarchy** displayed graphically; Lime supports single subclassing.
- **Type Hierarchy** presented as an indented list; Lime supports multiple subclassing.
- **Attribute** with all non-private attributes defined in the current class.
- **Inherited Attribute** with all attributes inherited from superclasses and supertypes.
- **Method** contains all methods defined in the current class. It gives the method signature and the precondition and postcondition defined in the current class. If the method redefines or implements a method of a supertype, it also gives the precondition and postcondition defined in supertypes. These are extracted from the Java *class* files of all supertypes.
- **Inherited Method** contains all inherited methods. It gives the method signature, precondition and postcondition.

5. OUTLOOK

Currently the development is still in an experimental stage. An exception handling mechanism needs to be integrated and the specification language needs to be extended with abstract date types. Currently specifications can only use the data types of the programming language.

6. REFERENCES

[1] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with assertions. In K. Havelund and G. Rosu, editors, *Proceedings of the First Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science, July 2001.

[2] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03), Electronic Notes in Theoretical Computer Science*, volume 66, pages 1–17, Trondheim, Norway, June 5–7, 2003. Elsevier Science.

[3] Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In *International Conference on Software Engineering Research and Practice (SERP)*, pages 322–328. Computer Science Research, Education, and Applications (CSREA) Press, Las Vegas, Nevada, USA, June 2002.

[4] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL '90)*, pages 125–135, San Francisco, January 1990. ACM Press. Addison-Wesley.

[5] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th international Conference on Software Engineering,*, pages 258–267, Berlin, Germany, March 1996. IEEE Computer Society Press.

[6] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *ACM SIGPLAN Notices , Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 37, pages 231–245, Seattle, Washington, USA, November 2002.

[7] R. Kramer. iContract - the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295–307, 1998.

[8] B. B. Kristensen, O. L. Madsen, B. Moeller-Pedersen, and K. Nygaard. The BETA programming language. In B. D. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[9] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[10] B. Meyer. *Object-Oriented Software Construction 2nd edition*. Prentice-Hall, 1997.

[11] A. L. Powell, J. C. French, and J. C. Knight. A systematic approach to creating and maintaining software documentation. In *Proceedings of the 1996 ACM symposium on Applied Computing*, pages 201–208, Philadelphia, Pennsylvania, February 1996.

[12] E. Sekerinski. Concurrent object-oriented programs: From specification to code. In *First International Symposium on Formal Methods for Components and Objects, FMCO 02*, Lecture Notes in Computer Science 2852, pages 403–423, Leiden, The Netherlands, 2003. Springer-Verlag.

[13] J. M. Wing. Writing Larch interface language specification. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.